Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
Algoritmos Greedy

18 de septiembre 2023

Tobías Serpa
108266

Carolina Mauro
108294

Valentina Adelsflugel
108201

1. Objetivo

El propósito de este trabajo práctico es consolidar nuestros conocimientos sobre los algoritmos Greedy, tema que hemos abordado en las primeras clases de la asignatura. Nuestra tarea consiste en primero analizar el problema brindado en el enunciado y luego conseguir su solución más óptima con un algoritmo Greedy.

Finalmente realizamos un profundo análisis sobre el mismo, como lo es estudiar detalladamente su complejidad, ponerlo a prueba con diferentes sets de datos planteados por la cátedra y elaborados por nosotros, examinar y generar sus métricas, y documentar todos los aspectos relevantes en este informe.

2. ¿Qué es un Algoritmo Greedy?

Son algoritmos que iterativamente aplican una regla sencilla que nos permite obtener en cada paso el óptimo local a nuestro problema actual. Esta sucesión de óptimos locales, nos logra llevar al óptimo global. Sin embargo, no siempre es así, por lo que plantear un algoritmo de este tipo es un proceso de análisis y de prueba y error hasta encontrar la regla que nos lleve a la solución óptima en todos los casos.

3. Análisis del problema

En esta ocasión debemos ayudar a Scaloni a analizar los próximos n rivales de la selección argentina. Para comenzar a pensar la solución, nuestro paso inicial fue definir las variables en cuestión:

$n \rightarrow$ Cantidad de compilados a analizar. Corresponde a la cantidad de rivales.

$s_i \rightarrow$ Tiempo que le lleva a Scaloni ver el compilado i $(1 \leq i \leq n)$

$a_i \rightarrow$ Tiempo que le lleva a un ayudante ver el compilado i

Un factor a tener en cuenta también, es que hay tantos ayudantes como compilados a analizar. Esto quiere decir que siempre habrá uno disponible al momento de necesitarlo.

Figura 1: Variables de nuestro problema.

Luego, planteamos diferentes criterios y aplicándolos a ejemplos de $n = 3$ fuimos descartando soluciones y detectando patrones que nos llevaron a nuestra solución final.

La pregunta central es: ¿En qué orden veo los videos? Tenemos que determinar cuál es el próximo rival que vamos a analizar una vez que finalizamos con uno.

A continuación procedemos a realizar una demostración de nuestro método de análisis:

- Comenzamos tomando como próximo video a analizar a aquel que le demande menos tiempo a Scaloni, sin importar los tiempos a_i. Entonces, por ejemplo, tomando un caso en el que $[(s_1, a_1), (s_2, a_2), (s_3, a_3)] = [(3, 3), (1, 8), (5, 1)]$, se deberían ver en este orden: $[(1, 8), (3, 3), (5, 1)]$. Por cada compilado calculamos el minuto exacto en el que se terminaría de revisar, esto corresponde a la siguiente cuenta:

```
(1, 8) -> 1 + 8 = termina en el minuto 9
(3, 3) -> (1 + 3) + 3 = termina en el minuto 7
(5, 1) -> (1 + 3 + 5) + 1 = termina en el minuto 10
```

  Es decir, por cada paso debemos sumar el tiempo que va tardando Scaloni y a eso agregarle el tiempo del ayudante para ese video. Esto es porque el DT no espera a que finalice el análisis del ayudante anterior, sino que se hace en paralelo.

  En este caso, se terminan de analizar todos los rivales en 10 minutos. En particular, nos dio la solución óptima pero, inmediatamente buscamos un contraejemplo: $[(s_1, a_1), (s_2, a_2), (s_3, a_3)] = [(1, 1), (3, 3), (5, 8)]$. Acá vemos que el tiempo total, realizando la cuenta correspondiente, nos da 17 minutos. Sin embargo, hay una solución mejor de 13 minutos en el orden $[(5, 8), (1, 1), (3, 3)]$.

  De esta manera, descartamos esta propuesta.

- En segundo lugar, nos propusimos tomar como próximo compilado a aquel que más tiempo le tome a Scaloni. Entonces, por ejemplo, tomando el caso del ítem anterior: $[(s_1, a_1), (s_2, a_2), (s_3, a_3)] = [(3, 3), (1, 8), (5, 1)]$, se deberían ver en este orden: $[(5, 1), (3, 3), (1, 8)]$. Por cada compilado calculamos el minuto exacto en el que se terminaría de revisar:

```
(5, 1) -> min 6
(3, 3) -> min 11
(1, 8) -> min 17

total = 17
```

  Vemos que no lo resuelve, el óptimo es 10. Descartamos.

  A este punto, nos comenzamos a hacer preguntas. ¿Depende el problema de los s_i? ¿Deberíamos mirar tanto los s_i como los a_i? Es así que planteamos una solución en la que teníamos en cuenta tanto los tiempos de Scaloni, como los de los Ayudantes:

- Relacionamos a_i con s_i de la forma $\frac{a_i}{s_i}$ y para el mismo ejemplo obtuvimos nuevamente una solución no óptima.

  Ahora sí, algo no estábamos viendo y era que **los tiempos de Scaloni no condicionan la solución** en absoluto. Procedemos a explicarlo: Supongamos que los análisis de los rivales sólo consisten en un análisis del DT. En este escenario ideal, tendríamos por ejemplo: $(s_1, s_2, s_3) = [2, 3, 5]$. Si los hacemos en el orden $i = 1, 2, 3$ el tiempo total es de $t = 2 + 3 + 5 = 10$, si tomamos los compilados con orden $i = 3, 1, 2$ tenemos $t = 5 + 2 + 3 = 10$. Es decir, no importa en qué orden, siempre $t = \sum_{i=1}^{3} s_i$ por lo que confirmamos nuestra suposición: nuestro problema sólo depende de los tiempos de los Ayudantes.

4. Solución al problema

Para explicar nuestra solución, vamos con un ejemplo. Supongamos que tenemos el siguiente vector de tiempos: $[(s_1, a_1), (s_2, a_2), (s_3, a_3)] = [(1, 8), (3, 3), (5, 1)]$. Ahora tomamos como próximo

al compilado con mayor a_i. En otras palabras, ordenamos el vector de mayor a menor a_i y ese es el orden en el que se deben ver los videos. Podemos verlo gráficamente en una línea de tiempo, donde lo azul corresponde a los tiempos de Scaloni y lo celeste a los tiempos de los Ayudantes:

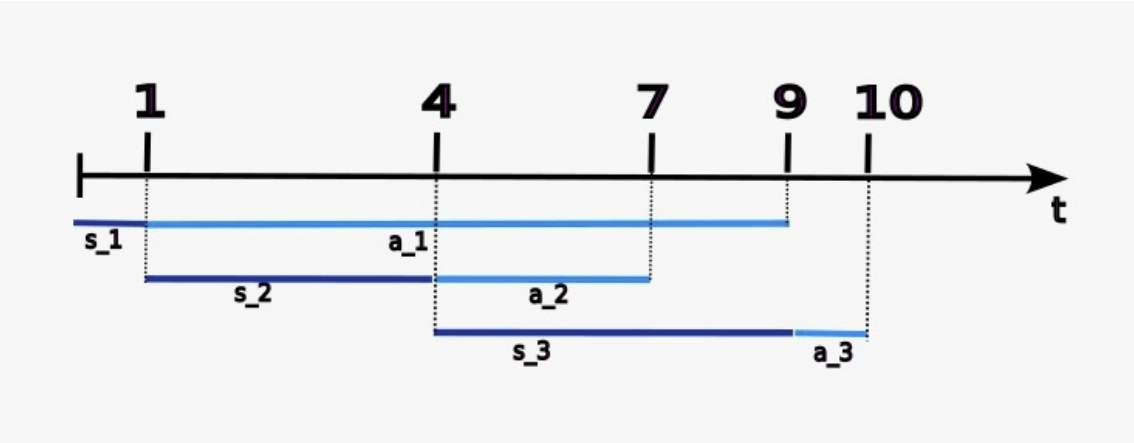


Figura 2: Ejemplo de solución.

En $t = 1m$ Scaloni ya analizó al rival número 1 e inmediatamente comienza a ver al rival 2. Simultáneamente, el Ayudante 1 revisa el compilado 1, y así sucesivamente.

El tiempo total es de 10 minutos. Vemos que el rival que más tiempo lleva analizar que es el primero, se lleva a cabo en paralelo con los demás. Este es el punto importante, nos conduce a que nuestro algoritmo sea óptimo, porque si lo hiciéramos al final, nos retrasaría todo dado que tendríamos que esperar a_1 minutos después de que todos ya hayan terminado.

¿Por qué nuestro algoritmo es Greedy?

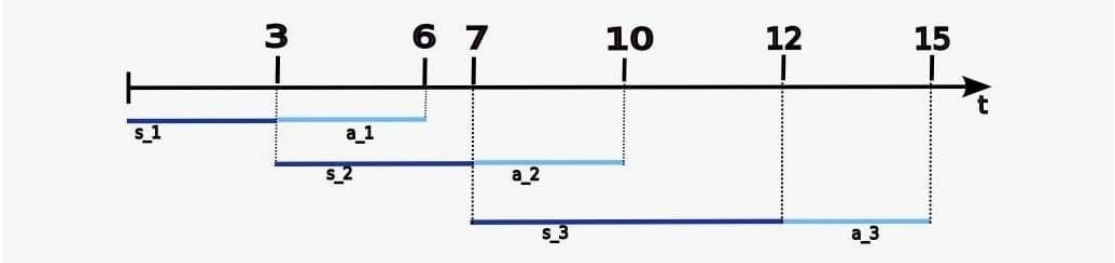
En principio podemos identificar que en cada paso estamos aplicando una regla sencilla: **analizar el rival con mayor tiempo de ayudante**.

Con este criterio tomamos decisiones en función de la información disponible en cada paso y buscamos el óptimo local, que en este caso resulta ser también el óptimo global.

¿Por qué es la solución óptima?

La forma más secilla de verlo es con el ejemplo de la Figura 2. Con este enfoque logramos compensar los tiempos largos de los ayudantes gracias a la simultaneidad de los análisis. Pero, ¿qué sucede en otros casos? Supongamos que los a_i son todos iguales. Esta es una situación análoga a aquella en la que no existen los Ayudantes, porque no condicionan el tiempo total. Veamos un ejemplo: Si $[(s_1, a_1), (s_2, a_2), (s_3, a_3)] = [(3, 3), (4, 3), (5, 3)]$ tenemos que:

- Al ordenar por a_i decreciente, el arreglo ya está ordenado (son todos iguales).
- El tiempo en el que se finaliza el análisis es de 15 minutos:



Si variamos el orden, es decir, calculamos el tiempo para todas las posibles combinaciones, vemos que no cambia:

```python
#Función para calcular por fuerza bruta cual es el óptimo
from itertools import permutations

def calcular_optimo(n, s_i, a_i):
    permutations_indices = permutations(range(n))

    tiempo_minimo = float('inf')

    for perm_indices in permutations_indices:
        orden_rivales = [(s_i[i], a_i[i]) for i in perm_indices]
        tiempo = tiempo_total(orden_rivales)
        print("Tiempo total: ", tiempo)
        if tiempo < tiempo_minimo:
            tiempo_minimo = tiempo
    return tiempo_minimo

calcular_optimo(3, [3,4,5], [3, 3, 3])
```

✓ 0.0s

```
Tiempo total:  15
Tiempo total:  15
Tiempo total:  15
Tiempo total:  15
Tiempo total:  15
Tiempo total:  15
```

(Esta función será explicada en la sección siguiente).

Un ultimo escenario posible sería que los tiempos de Scaloni sean todos iguales. Esto no nos modifica en absoluto nuestro problema por la afirmación que ya hicimos previamente acerca de los s_i .

Por lo tanto podemos concluír, que para todos los posibles casos, nuestro algoritmo nos da la mejor solución.

4.1. Implementación del algoritmo

Decidimos trabajar con el lenguaje de programación Python dado que es muy amigable con el manejo de arrays, gráficos y además es fácil de entender. Armamos una notebook jupyter para documentar mejor el código, que está organizada de la siguiente manera:

- Una sección inicial donde definimos funciones genéricas para: leer los archivos de prueba y los resultados esperados, ordenar los tiempos según un criterio y calcular el tiempo total del análisis. También hicimos una función que por fuerza bruta, es decir, calculando el tiempo para todas las posibles combinaciones de orden, nos devuelve el tiempo óptimo. Esta última la utilizamos para definir para nuestros sets de datos cuál es el óptimo, para que luego utilizando nuestro algoritmo Greedy podamos confirmar si nos daba el resultado correcto.
- Las secciones siguientes corresponden a todas las soluciones propuestas. Explicamos por qué algunas fallaron y finalmente presentamos la óptima.

Para la representación de los elementos s y a para cada i , utilizamos tuplas, y con arreglos de esas tuplas, representamos una instancia de nuestro problema.

Nuestro algoritmo consiste en dos pasos:

1. Ordenar el vector de tiempos con el criterio que llamamos `sort_by_ai`, es decir tomando la segunda componente de cada tupla como clave y ordeando de mayor a menor.
2. Calcular el tiempo óptimo. Para esto, debemos recorrer todo el arreglo de tiempos y por cada uno:
   a) Sumar el tiempo s_i a un contador del tiempo total de Scaloni.
   b) Sumar el tiempo total de Scaloni con el tiempo a_i y agregarlo a un vector (`tiempos_por_video`).
3. Finalmente calcular el máximo del arreglo `tiempos_por_video`.

Veamos el código:

```
def ordenar_tupla(criterio, tiempos, reverse = False):
    tiempos.sort(key = criterio, reverse = reverse)
    return tiempos
```

Función que ordena los tiempos.

Esta función recibe un criterio de ordenamiento que puede ser:

```
sort_by_ai = lambda x: x[1]
sort_by_si = lambda x: x[0]
sort_by_ai_si = lambda x: x[1]/x[0]
```

En cada sección, utilizamos el criterio correspondiente a la solución planteada. El correspondiente a la óptima es el de sort_by_ai.

Por otro lado, tenemos el algoritmo que nos calcula el tiempo total que nos lleva analizar a los rivales dado un vector de tuplas de s_i y a_i:

```
def tiempo_total(tiempos):
    total_scaloni = 0
    tiempos_por_video = []
    for tupla in tiempos:
        total_scaloni += tupla[0]
        tiempos_por_video.append(total_scaloni + tupla[1])
    return max(tiempos_por_video)
```

Función que calcula el tiempo de análisis total.

Un ejemplo de uso se puede ver en la sección 4.3 de este informe.

5. Complejidad del algoritmo

En esta sección, analizaremos las complejidades de algunos algoritmos que resultan esenciales para la resolución del problema planteado, como así también lo haremos para nuestra solución final.

5.1. Tiempo total

Esta función, fundamental para nuestra solución ya que calcula el timepo que lleva que todos los vídeos sean vistos por Scaloni y un ayudante, posee una complejidad temporal igual a O(n). Esto es así debido al ciclo *for* que recorre todas las tuplas dentro del vector de tiempos. Dentro del bucle todas las operaciones son de tiempo constante, son una suma y un agregado de un elemento al final de un arreglo. Por último se calcula el máximo de un array, esto es también O(n).

Es así que el tiempo de ejecución del algoritmo depende exclusivamente de la longitud del vector de tiempos, siendo O(n).

```python
def tiempo_total(tiempos):
    total_scaloni = 0
    tiempos_por_video = []
    for tupla in tiempos:
        total_scaloni += tupla[0]
        tiempos_por_video.append(total_scaloni + tupla[1])
    return max(tiempos_por_video)
```

5.2. Ordenar tupla

Este algoritmo, que es utilizado para ordenar los tiempos que tardan Scaloni y sus ayudantes en revisar los videos de los rivales, posee una complejidad temporal de O(n log n). Sin embargo, en el mejor de los casos, donde la lista ya está ordenada, la complejidad del tiempo se reduce a O(n). Esta complejidad se debe a la función sort de Python utilizada para ordenar los tiempos.

```python
def ordenar_tupla(criterio, tiempos, reverse = False):
    tiempos.sort(key = criterio, reverse = reverse)
    return tiempos
```

5.3. Ejemplo de solución

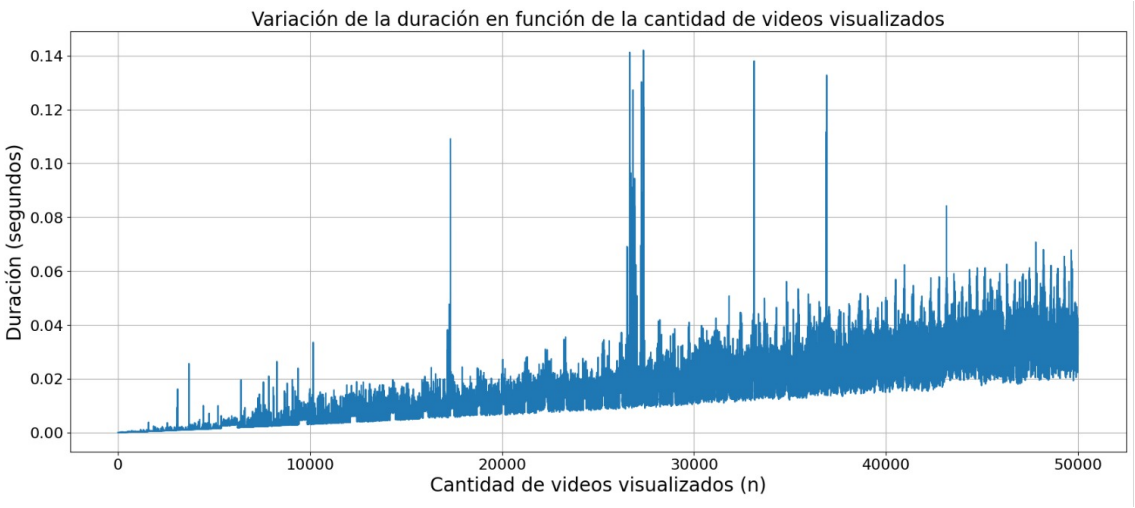
Como podemos notar en nuestra solución, primero realizamos un ordenamiento de los tiempos con el criterio definido anteriormente, y luego calculamos los tiempos totales. Dado que la parte más costosa del algoritmo es el ordenamiento de las listas, la complejidad temporal total del algoritmo se verá dominada por esta. Es por esto que la complejidad es O(n log n).

```python
tiempos_n3_ordenados = ordenar_tupla(sort_by_ai, tiempos_n3, True)
tiempo_n3_total = tiempo_total(tiempos_n3_ordenados)

tiempos_n10_ordenados = ordenar_tupla(sort_by_ai, tiempos_n10, True)
tiempo_n10_total = tiempo_total(tiempos_n10_ordenados)

tiempos_n100_ordenados = ordenar_tupla(sort_by_ai, tiempos_n100, True)
tiempo_n100_total = tiempo_total(tiempos_n100_ordenados)

tiempos_n10000_ordenados = ordenar_tupla(sort_by_ai, tiempos_n10000, True)
tiempo_n10000_total = tiempo_total(tiempos_n10000_ordenados)
```

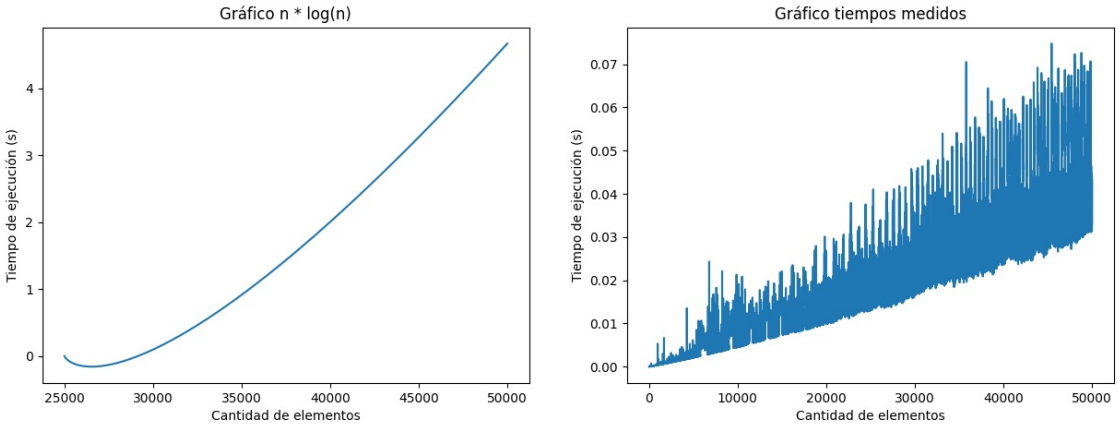
6. Mediciones

Se realizaron dos mediciones para visualizar la duración de los algoritmos propuestos.

Por un lado, para el primer gráfico, se genero una muestra aleatoria de 50.000 pares (s_i, a_i) y se midió el tiempo que tardó nuestro algoritmo en hallar la solución óptima sumando de a 1 elemento en cada iteración.

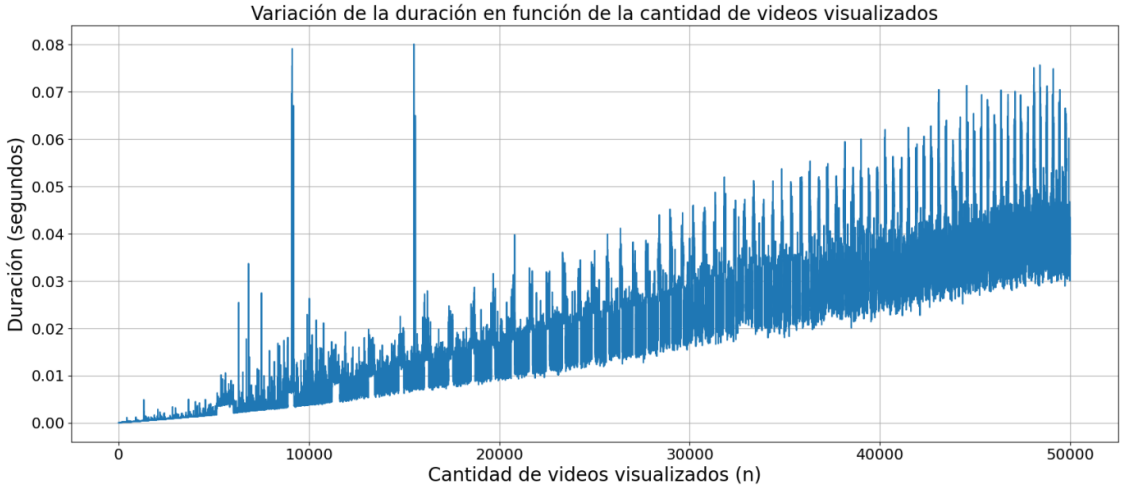


Según lo que analizamos, nuestro algoritmo presenta teóricamente una complejidad temporal O(n*log(n)). Para visualizar esto, graficamos la función n*log(n) junto con los tiempos medidos:



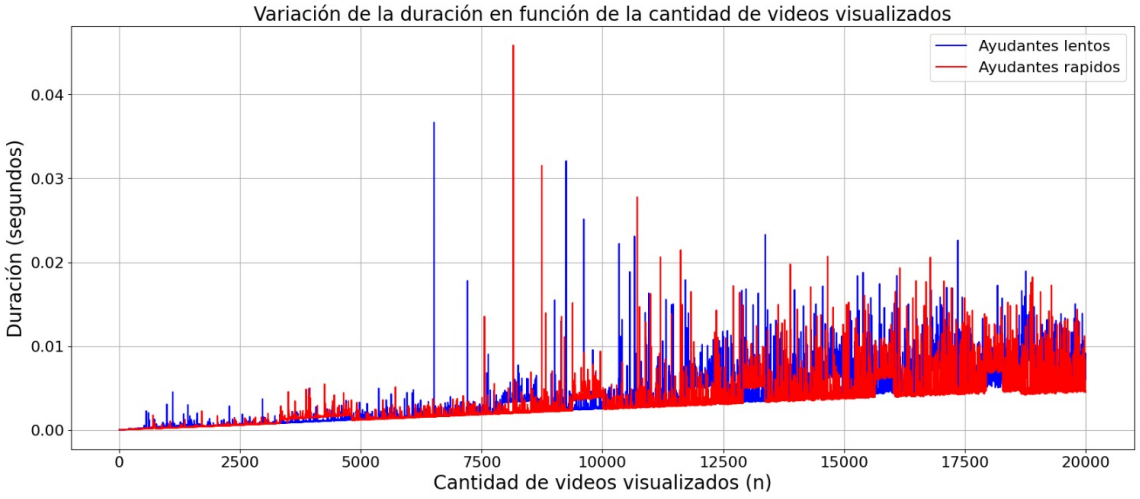
Vemos que presenta una similitud, pero no es exactamente igual dado que son datos experimentales y aproximados.

Realizamos otro gráfico con el mismo set de datos pero ordenando de manera ascendente los tiempos de los ayudantes. Esto vendría a representar el 'peor caso posible' para el algoritmo propuesto porque sería el orden inverso al deseado.



Como se puede observar, el peor caso se da cuando los tiempos de los ayudantes están ordenados de forma descentende en nuesto set de datos. Esto genera que nuestro algoritmo presente una complejidad temporal O(n*log(n)) como habíamos definido.

Por otro lado, el último gráfico muestra la comparación de dos muestras de datos (ambas con 10.000 datos). En la primera (azul) los tiempos que tarda Scaloni en analizar los videos varían entre 100-200m mientras que el tiempo que tardan los ayudantes entre 1-10m. Podría decirse que este es el caso donde tenemos un "Scaloni más lento que los ayudantes". En contraste, el segundo set de datos esta conformado de manera inversa tal que Scaloni tarda ente 1-10 minutos contra 100-200 que tardan los ayudantes. En este caso "Scaloni es más rápido que los ayudantes".



Este gráfico nos ayuda a poder comparar estas situaciones donde alguno de las dos partes es significativamente más rápida/lenta que la otra. Como podemos observar que el tiempo en el que tarda en hallar nuestro algorimo la solución más óptima es para ambos sets de datos bastante parecido.

Esto era algo de esperar ya que como habíamos notado anteriormente, el tiempo de Scaloni no tiene inferencia en el tiempo en el que se tarda en ver los videos de la forma mas óptima. Por lo tanto, nuestro algoritmo exhibe un tiempo de ejecución no dependiente del escenario (sea uno de las dos partes más rápida"que la otra") pero si de la cantidad de videos a analizar.

7. Conclusiones

El objetivo del presente trabajo práctico era hallar un algoritmo que encuentre la solución óptima al problema dado. Esta búsqueda fue planteada pensando en que nuestro problema se podía resolver con un algoritmo Greedy.

Como se detalló en la sección de Análisis, nosotros contabamos con dos variables que inferian en la solución: los tiempos que le tomaba a Scaloni y los tiempos que le tomaba a los ayudantes ver los videos. Al principio pensamos que ambos iban a influír en la búsqueda de la solución más óptima. Por esta razón planteamos inicialmente soluciones en las que teníamos en cuenta el tiempo que le demoraba a ambas partes, ya que creíamos que de esta manera ibamos a poder encontrar un óptimo global.

Al notar que no estabamos obteniendo el resultado esperado ya que limitamos nuestro pensamiento a obtener resultados inmediatos, decidimos cambiar el foco. Pudimos concluir de esta forma que Scaloni no condicionaba en absoluto la solución y que nuestro algoritmo debía únicamente centrarse en los tiempos de los ayudantes.

Esto se da así ya que la solución más óptima siempre va a ser, como mínimo la suma de todos los tiempos que le lleve a Scaloni ver los videos. Entonces, como los ayudantes ven el video i una vez Scaloni lo haya analizado, nuestro problema se redució a encontrar la mejor forma en que los ayudantes vean los videos.

Pudimos de esta forma identificar la mejor estrategia para lograr llegar al óptimo global. Esto fue posible gracias a que logramos entender la dinámica y propuestas de los algoritmos Greedy, permitiendos obtener una solución óptima.

Respecto al tiempo de ejecución de nuestra solución, nos parece interesante recalcar que se nota un aumento notable del mismo cuando aumenta la cantidad de videos a analizar.